A Next-Generation Platform for Analyzing Executables[⋆]

T. Reps[1,2], G. Balakrishnan[1], J. Lim[1], and T. Teitelbaum[2]

[1] Comp. Sci. Dept., University of Wisconsin; {reps,bgogul,junghee}@cs.wisc.edu
[2] GrammaTech, Inc.; {tt}@grammatech.com

Abstract. In recent years, there has been a growing need for tools that an analyst can use to understand the workings of COTS components, plugins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. Static analysis provides techniques that can help with such problems; however, there are several obstacles that must be overcome:

- For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. Even if it is present, it cannot be relied upon.
- To understand memory-access operations, it is necessary to determine the set of addresses accessed by each operation. This is difficult because
  - While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (difficult).
  - Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  - There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.
  - Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.

We have developed static-analysis algorithms to recover information about the contents of memory locations and how they are manipulated by an executable. By combining these analyses with facilities provided by the IDAPro and CodeSurfer toolkits, we have created CodeSurfer/x86, a prototype tool for browsing, inspecting, and analyzing x86 executables. From an x86 executable, CodeSurfer/x86 recovers intermediate representations that are similar to what would be created by a compiler for a program written in a high-level language. CodeSurfer/x86 also supports a scripting language, as well as several kinds of sophisticated pattern-matching capabilities. These facilities provide a platform for the development of additional tools for analyzing the security properties of executables.

1 Introduction

Market forces are increasingly pushing companies to deploy COTS software when possible—for which source code is typically unavailable—and to outsource development when custom software is required. Moreover, a great deal of legacy code—for which design documents are usually out-of-date, and for which source code is sometimes unavailable and sometimes non-existent—will continue to be left deployed. An important challenge during the coming decade will be how to identify bugs and security vulnerabilities in such systems. Methods are needed to determine whether third-party and legacy application programs can perform malicious operations (or can be induced to perform malicious operations), and to be able to make such judgments in the absence of source code.

[⋆] Portions of this paper have appeared in [3, 4].

Recent research in programming languages, software engineering, and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities [25, 40, 20, 14, 8, 5, 10, 27, 17, 9]. In these tools, static analysis is used to determine a conservative answer to the question "Can the program reach a bad state?"[3] In principle, such tools would be of great help to an analyst trying to detect malicious code hidden in software, except for one important detail: the aforementioned tools all focus on analyzing *source code* written in a high-level language. Even if source code were available, there are a number of reasons why analyses that start from source code do not provide the right level of detail for checking certain kinds of properties, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. (See §2.)

In contrast, our work addresses the problem of finding bugs and security vulnerabilities in programs when source code is unavailable. Our goal is to create a platform that carries out static analysis on executables and provides information that an analyst can use to understand the workings of potentially malicious code, such as COTS components, plugins, mobile code, and DLLs, as well as memory snapshots of worms and virus-infected code. A second goal is to use this platform to create tools that an analyst can employ to determine such information as

– whether a program contains inadvertent security vulnerabilities
– whether a program contains deliberate security vulnerabilities, such as back doors, time bombs, or logic bombs. If so, the goal is to provide information about activation mechanisms, payloads, and latencies.

We have developed a tool, called CodeSurfer/x86, that serves as a prototype for a next-generation platform for analyzing executables. CodeSurfer/x86 provides a security analyst with a powerful and flexible platform for investigating the properties and possible behaviors of an x86 executable. It uses static analysis to recover intermediate representations (IRs) that are similar to those that a compiler creates for a program written in a high-level language. An analyst is able to use (i) CodeSurfer/x86's GUI, which provides mechanisms to understand a program's chains of data and control dependences, (ii) CodeSurfer/x86's scripting language, which provides access to all of the intermediate representations that CodeSurfer/x86 builds, and (iii) GrammaTech's Path Inspector, which is a model-checking tool that uses a sophisticated pattern-matching engine to answer questions about the flow of execution in a program.

Because CodeSurfer/x86 was designed to provide a platform that an analyst can use to understand the workings of potentially malicious code, a major challenge is that the tool must assume that the x86 executable is untrustworthy, and hence symbol-table and debugging information cannot be relied upon (even if it is present). The algorithms used in CodeSurfer/x86 provide ways to meet this challenge.

Although the present version of CodeSurfer/x86 is targeted to x86 executables, the techniques used [3, 34, 37, 32] are language-independent and could be applied to other types of executables. In addition, it would be possible to extend CodeSurfer/x86 to

[3] Static analysis provides a way to obtain information about the possible states that a program reaches during execution, but without actually running the program on specific inputs. Static-analysis techniques explore the program's behavior for *all* possible inputs and *all* possible states that the program can reach. To make this feasible, the program is "run in the aggregate"—i.e., on descriptors that represent *collections* of memory configurations [15].

use symbol-table and debugging information in situations where such information is available and trusted—for instance, if you have the source code for the program, you invoke the compiler yourself, and you trust the compiler to supply correct symbol-table and debugging information. Moreover, the techniques extend naturally if source code is available: one can treat the executable code as just another IR in the collection of IRs obtainable from source code. The mapping of information back to the source code would be similar to what C source-code tools already have to perform because of the use of the C preprocessor (although the kind of issues that arise when debugging optimized code [26, 43, 16] complicate matters).

The remainder of paper is organized as follows: §2 illustrates some of the advantages of analyzing executables. §3 describes CodeSurfer/x86. §4 gives an overview of the model-checking facilities that have been coupled to CodeSurfer/x86. §5 discusses related work.

2 Advantages of Analyzing Executables

This section discusses why an analysis that works on executables can provide more accurate information than an analysis that works on source code.[4] An analysis that works on source code can fail to detect certain bugs and vulnerabilities due to the WYSINWYX phenomenon: "What You See Is Not What You eXecute" [4], which can cause there to be a mismatch between what a programmer intends and what is actually executed by the processor. The following source-code fragment, taken from a login program, illustrates the issue [29]:

```
memset(password, ‘\0’, len);
free(password);
```

The login program temporarily stores the user's password—in clear text—in a dynamically allocated buffer pointed to by the pointer variable `password`. To minimize the lifetime of the password, which is sensitive information, the code fragment shown above zeroes-out the buffer pointed to by `password` before returning it to the heap. Unfortunately, a compiler that performs useless-code elimination may reason that the program never uses the values written by the call on `memset`, and therefore the call on memset can be removed—thereby leaving sensitive information exposed in the heap. This is not just hypothetical; a similar vulnerability was discovered during the Windows security push in 2002 [29]. This vulnerability is invisible in the source code; it can only be detected by examining the low-level code emitted by the optimizing compiler.

A second example where analysis of an executable does better than typical source-level analyses involves pointer arithmetic and an indirect call:

```
int (*f)(void);
int diff = (char*)&f2 - (char*)&f1; // The offset between f1 and f2
f = &f1;
f = (int (*)())((char*)f + diff); // f now points to f2
(*f)(); // indirect call;
```

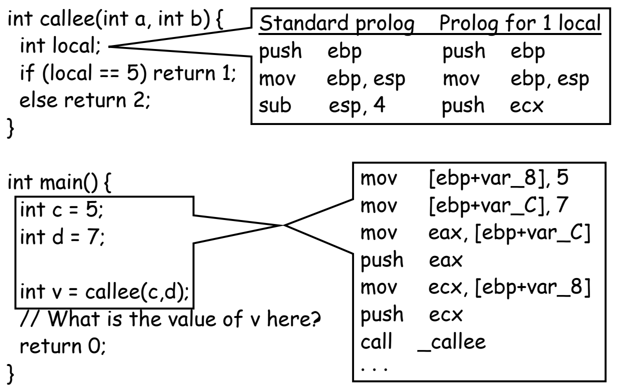
Existing source-level analyses (that we know of) are ill-prepared to handle the above code. The conventional assumption is that arithmetic on function pointers leads to undefined behavior, so source-level analyses either (a) assume that the indirect function

[4] Terms like "an analysis that works on source code" and "source-level analyses" are used as a shorthand for "analyses that work on IRs built from the source code."

call might call any function, or (b) ignore the arithmetic operations and assume that the indirect function call calls `f1` (on the assumption that the code is ANSI-C compliant). In contrast, the analysis described by Balakrishnan and Reps [3] correctly identifies `f2` as the invoked function. Furthermore, the analysis can detect when arithmetic on addresses creates an address that does not point to the beginning of a function; the use of such an address to perform a function "call" is likely to be a bug (or else a very subtle, deliberately introduced security vulnerability).

A third example involves a function call that passes fewer arguments than the procedure expects as parameters. (Many compilers accept such (unsafe) code as an easy way of implementing functions that take a variable number of parameters.) With most compilers, this effectively means that the call-site passes some parts of one or more local variables of the calling procedure as the remaining parameters (and, in effect, these are passed by reference—an assignment to such a parameter in the callee will overwrite the value of the corresponding local in the caller.) An analysis that works on executables can be created that is capable of determining what the extra parameters are [3], whereas a source-level analysis must either make a cruder over-approximation or an unsound under-approximation.

A final example is shown in Fig. 1. The C code on the left uses an uninitialized variable (which triggers a compiler warning, but compiles successfully). A source-code analyzer must assume that `local` can have any value, and therefore the value of `v` in `main` is either 1 or 2. The assembly listings on the right show how the C code could be compiled, including two variants for the prolog of function `callee`. The Microsoft compiler (cl) uses the second variant, which includes the following strength reduction:



```
int callee(int a, int b) {
  int local;
  if (local == 5) return 1;
  else return 2;
}

int main() {
  int c = 5;
  int d = 7;

  int v = callee(c,d);
  // What is the value of v here?
  return 0;
}
```

```
Standard prolog    Prolog for 1 local
push   ebp         push   ebp
mov    ebp, esp    mov    ebp, esp
sub    esp, 4      push   ecx
```

```
mov    [ebp+var_8], 5
mov    [ebp+var_C], 7
mov    eax, [ebp+var_C]
push   eax
mov    ecx, [ebp+var_8]
push   ecx
call   _callee
. . .
```

Fig. 1. Example of unexpected behavior due to compiler optimization. The box at the top right shows two variants of code generated by an optimizing compiler for the prolog of `callee`. Analysis of the second of these reveals that the variable `local` necessarily contains the value 5.

The instruction `sub esp,4` *that allocates space for* `local` *is replaced by a* `push` *instruction of an arbitrary register (in this case,* `ecx`*).*

An analysis of the executable can determine that this optimization results in `local` being initialized to 5, and therefore `v` in `main` can only have the value 1.

To summarize, the advantage of an analysis that works on executables is that an executable contains the actual instructions that will be executed, and hence provides information that reveals the actual behavior that arises during program execution. This information includes

– memory-layout details, such as (i) the positions (i.e., offsets) of variables in the runtime stack's activation records, and (ii) padding between structure fields.

– register usage
– execution order (e.g., of actual parameters)
– optimizations performed
– artifacts of compiler bugs

Access to such information can be crucial; for instance, many security exploits depend on platform-specifi c features, such as the structure of activation records. Vulnerabilities can escape notice when a tool does not have information about adjacency relationships among variables.

In contrast, there are a number of reasons why analyses based on source code do not provide the right level of detail for checking certain kinds of properties:

– Source-level tools are only applicable when source is available, which limits their usefulness in security applications (e.g., to analyzing code from open-source projects).
– Analyses based on source code typically make (unchecked) assumptions, e.g., that the program is ANSI-C compliant. This often means that an analysis does not account for behaviors that are allowed by the compiler (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of arrays and are subsequently dereferenced; etc.)
– Programs typically make extensive use of libraries, including dynamically linked libraries (DLLs), which may not be available in source-code form. Typically, source-level analyses are performed using code stubs that model the effects of library calls. Because these are created by hand they are likely to contain errors, which may cause an analysis to return incorrect results.
– Programs are sometimes modifi ed subsequent to compilation, e.g., to perform optimizations or insert instrumentation code [41]. (They may also be modifi ed to insert malicious code.) Such modifi cations are not visible to tools that analyze source.
– The source code may have been written in more than one language. This complicates the life of designers of tools that analyze source code because multiple languages must be supported, each with its own quirks.
– Even if the source code is primarily written in one high-level language, it may contain inlined assembly code in selected places. Source-level analysis tools typically either skip over inlined assembly code [13] or do not push the analysis beyond sites of inlined assembly code [1].

Thus, even if source code is available, a substantial amount of information is hidden from analyses that start from source code, which can cause bugs, security vulnerabilities, and malicious behavior to be invisible to such tools. Moreover, a source-level analysis tool that strives to have greater fi delity to the program that is actually executed would have to duplicate all of the choices made by the compiler and optimizer; such an approach is doomed to failure.

3 Analyzing Executables in the Absence of Source Code

To be able to apply techniques like the ones used in [25, 40, 20, 14, 8, 5, 10, 27, 17, 9], one already encounters a challenging program-analysis problem. From the perspective of the compiler community, one would consider the problem to be "IR recovery": one needs to recover *intermediate representations* from the executable that are similar to those that would be available had one started from source code. From the perspective of the model-checking community, one would consider the problem to be that of "model extraction": one needs to extract a suitable *model* from the executable. To solve the IR-recovery problem, several obstacles must be overcome:

- For many kinds of potentially malicious programs, symbol-table and debugging information is entirely absent. Even if it is present, it cannot be relied upon.
- To understand memory-access operations, it is necessary to determine the set of addresses accessed by each operation. This is diffi cult because
  - While some memory operations use explicit memory addresses in the instruction (easy), others use indirect addressing via address expressions (diffi cult).
  - Arithmetic on addresses is pervasive. For instance, even when the value of a local variable is loaded from its slot in an activation record, address arithmetic is performed.
  - There is no notion of type at the hardware level, so address values cannot be distinguished from integer values.
  - Memory accesses do not have to be aligned, so word-sized address values could potentially be cobbled together from misaligned reads and writes.

For the past few years, we have been working to create a prototype next-generation platform for analyzing executables. The tool set that we have developed extends static vulnerability-analysis techniques to work directly on executables, even in the absence of source code. The tool set builds on (i) recent advances in static analysis of program executables [3], and (ii) new techniques for software model checking and dataflow analysis [7, 36, 37, 32]. The main components of the tool set are *CodeSurfer/x86*, *WPDS++*, and the *Path Inspector*:

- CodeSurfer/x86 recovers IRs from an executable that are similar to the IRs that source-code-analysis tools create—but, in many respects, the IRs that CodeSurfer/x86 builds are more precise. CodeSurfer/x86 also provides an API to these IRs.
- WPDS++ [31] is a library for answering generalized reachability queries on *weighted pushdown systems* (WPDSs) [7, 36, 37, 32]. This library provide a mechanism for defi ning and solving model-checking and dataflow-analysis problems. To extend CodeSurfer/x86's analysis capabilities, the CodeSurfer/x86 API can be used to extract a WPDS model from an executable and to run WPDS++ on the model.
- The Path Inspector is a software model checker built on top of CodeSurfer and WPDS++. It supports safety queries about a program's possible control confi gurations.

In addition, by writing scripts that traverse the IRs that CodeSurfer/x86 recovers, the tool set can be extended with further capabilities (e.g., decompilation, code rewriting, etc.).

Fig. 2 shows how these components fi t together. CodeSurfer/x86 makes use of both IDAPro [30], a disassembly toolkit, and GrammaTech's CodeSurfer system [13], a toolkit originally developed for building program-analysis and inspection tools that analyze source code. These components are glued together by a piece called the Connector, which uses two static analyses—value-set analysis (VSA) [3] and aggregate-structure identifi cation (ASI) [34] to recover information about the contents of memory locations and how they are manipulated by an executable.[5]

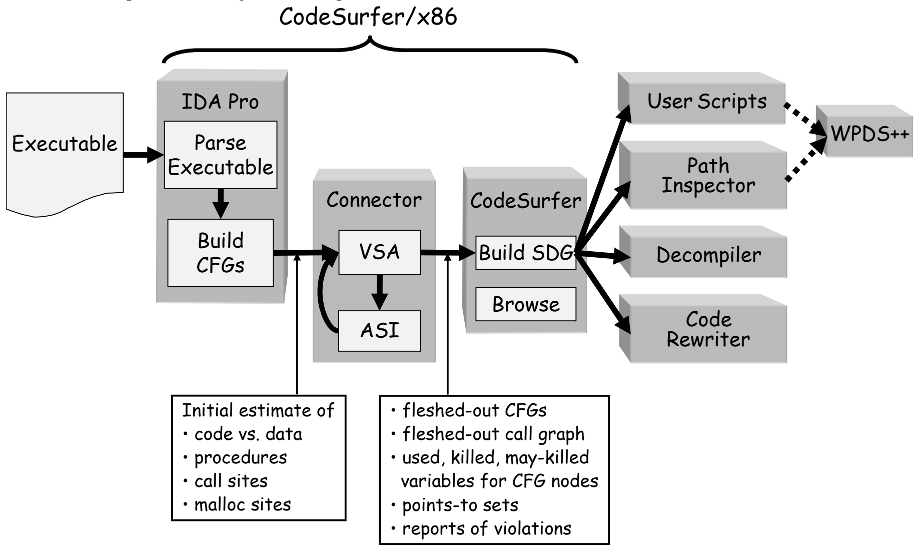


Fig. 2. Organization of CodeSurfer/x86 and companion tools.

An x86 executable is fi rst disassembled using IDAPro. In addition to the disassembly listing, IDAPro also provides access to the following information:

Statically known memory addresses and offsets: IDAPro identifi es the statically known memory addresses and stack offsets in the program, and renames all occurrences of these quantities with a consistent name. This database is used to defi ne the set of data objects in terms of which (the initial run of) VSA is carried out; these objects are called *a-locs*, for “abstract locations”. VSA is an analysis that, for each instruction, determines an over-approximation of the set of values that each a-loc could hold.

Information about procedure boundaries: X86 executables do not have information about procedure boundaries. IDAPro identifi es the boundaries of most of the procedures in an executable.[6]

Calls to library functions: IDAPro discovers calls to library functions using an algorithm called Fast Library Identifi cation and Recognition Technology (FLIRT) [23].

IDAPro provides access to its internal resources via an API that allows users to create plug-ins to be executed by IDAPro. CodeSurfer/x86 uses a plug-in to IDAPro, called the Connector, that creates data structures to represent the information that it obtains from IDAPro (see Fig. 2); VSA and ASI are implemented using the data structures

[5] VSA also makes use of the results of an additional static-analysis phase, called affine-relation analysis (ARA), which, for each program point, identifies affine relationships [33] that hold among the values of registers; see [3, 32].

[6] IDAPro does not identify the targets of all indirect jumps and indirect calls, and therefore the call graph and control-fbw graphs that it constructs are not complete. However, the information computed during VSA is used to augment the call graph and control-fbw graphs on-the-fly to account for indirect jumps and indirect calls.

created by the Connector. The IDAPro/Connector combination is also able to create the same data structures for DLLs, and to link them into the data structures that represent the program itself. This infrastructure permits whole-program analysis to be carried out—including analysis of the code for all library functions that are called.

CodeSurfer/x86 makes no use of symbol-table or debugging information. Instead, the results of VSA and ASI provide a substitute for absent or untrusted symbol-table and debugging information. Initially, the set of a-locs is determined based on the static memory addresses and stack offsets that are used in instructions in the executable. Each run of ASI refines the set of a-locs used for the next run of VSA.

Because the IRs that CodeSurfer/x86 recovers are extracted directly from the executable code that is run on the machine, and because the entire program is analyzed—including any libraries that are linked to the program—this approach provides a "higher fidelity" platform for software model checking than the IRs derived from source code that other software model checkers use [25, 40, 20, 14, 8, 5, 10, 27, 17, 9].

CodeSurfer/x86 supports a scripting language that provides access to all of the IRs that CodeSurfer/x86 builds for the executable. This provides a way to connect CodeSurfer/x86 to other analysis tools, such as model checkers (see §4), as well as to implement other tools on top of CodeSurfer/x86, such as decompilers, code rewriters, etc. It also provides an analyst with a mechanism to develop any additional "one-off" analyses he needs to create.

3.1 Memory-Access Analysis in the Connector

The analyses in CodeSurfer/x86 are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro. At the technical level, CodeSurfer/x86 addresses the following problem:

Given a stripped executable E, identify the

- procedures, data objects, types, and libraries that it uses

and

- for each instruction I in E and its libraries
- for each interprocedural calling context of I
- for each machine register and a-loc A

statically compute an accurate over-approximation to

- the set of values that A may contain when I executes
- the instructions that may have defined the values used by I
- the instructions that may use the values defined by execution of I

and provide effective means to access that information both interactively and under program control.

Value-Set Analysis. VSA [3] is a combined numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses (or *value set*) that each a-loc holds at each program point. The information computed during VSA is used to augment the call graph and control-flow graphs on-the-fly to account for indirect jumps and indirect function calls.

VSA is related to pointer-analysis algorithms that have been developed for programs written in high-level languages, which determine an over-approximation of the set of variables whose addresses each pointer variable can hold:

VSA determines an over-approximation of the set of addresses that each data object can hold at each program point.

At the same time, VSA is similar to range analysis and other numeric static-analysis algorithms that over-approximate the integer values that each variable can hold:

VSA determines an over-approximation of the set of integer values that each data object can hold at each program point.

The following insights shaped the design of VSA:

- A *non-aligned access* to memory—e.g., an access via an address that is not aligned on a 4-byte word boundary—spans parts of two words, and provides a way to forge a new address from *parts* of old addresses. It is important for VSA to discover information about the alignments and strides of memory accesses, or else most indirect-addressing operations appear to be possibly non-aligned accesses.
- To prevent most loops that traverse arrays from appearing to be possible stack-smashing attacks, the analysis needs to use relational information so that the values of a-locs assigned to within a loop can be related to the values of the a-locs used in the loop's branch condition (see [3, 33, 32]).
- It is desirable for VSA to track integer-valued and address-valued quantities *simultaneously*. This is crucial for analyzing executables because
  - integers and addresses are indistinguishable at execution time, and
  - compilers use address arithmetic and indirect addressing to implement such features as pointer arithmetic, pointer dereferencing, array indexing, and accessing structure fi elds.

  Moreover, information about integer values can lead to improved tracking of address-valued quantities, and information about address values can lead to improved tracking of integer-valued quantities.

VSA produces information that is more precise than that obtained via several more conventional numeric analyses used in compilers, including constant propagation, range analysis, and integer-congruence analysis. At the same time, VSA provides an analog of pointer analysis that is suitable for use on executables.

Aggregate-Structure Identification. One of the major stumbling blocks in analysis of executables is the diffi culty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays). CodeSurfer/x86 uses an iterative strategy for recovering such information; with each round, it refi nes its notion of the program's variables and types.

Initially, VSA uses a set of variables ("a-locs") that are obtained from IDAPro. Because IDAPro has relatively limited information available at the time that it applies its variable-discovery heuristics (i.e., it only knows about statically known memory addresses and stack offsets), what it can do is rather limited, and generally leads to a very coarse-grained approximation of the program's variables.

Once a given run of VSA completes, the value-sets for the a-locs at each instruction provide a way to identify an over-approximation of the memory accesses performed at that instruction. This information is used to refi ne the current set of a-locs by running a variant of the ASI algorithm [34], which identifi es commonalities among accesses to different parts of an aggregate data value. ASI was originally developed for analysis of Cobol programs: in that context, ASI ignores all of the type declarations in the program, and considers an aggregate to be merely a sequence of bytes of a given length; an aggregate is then broken up into smaller parts depending upon how the aggregate is accessed by the program. In the context in which we use ASI—namely, analysis of x86 executables—ASI cannot be applied until the results of VSA are already in hand: ASI requires points-to, range, and stride information to be available; however, for an x86 executable this information is not available until after VSA has been run.

ASI exploits the information made available by VSA (such as the values that a-locs can hold, sizes of arrays, and iteration counts for loops), which generally leads to a much more accurate set of a-locs than the initial set of a-locs discovered by IDAPro. For instance, consider a simple loop, implemented in source code as

```
int a[10], i;
for (i = 0; i < 10; i++)
   a[i] = i;
```

From the executable, IDAPro will determine that there are two variables, one of size 4 bytes and one of size 40 bytes, but will provide no information about the substructure of the 40-byte variable. In contrast, in addition to the 4-byte variable, ASI will correctly identify that the 40 bytes are an array of ten 4-byte quantities.

The Connector uses a refinement loop that performs repeated phases of VSA and ASI (see Fig. 2). The ASI results are used to refine the previous set of a-locs, and the refined set of a-locs is then used to analyze the program during the next round of VSA. The number of iterations is controlled by a command-line parameter.

ASI also provides information that greatly increases the precision with which VSA can analyze the contents of dynamically allocated objects (i.e., memory locations allocated using malloc or new). To see why, recall how the initial set of a-locs is identified by IDAPro. The a-loc abstraction exploits the fact that accesses to program variables in a high-level language are either complied into static addresses (for globals, and fields of struct-valued globals) or static stack-frame offsets (for locals and fields of struct-valued locals). However, fields of dynamically allocated objects are accessed in terms of offsets relative to the base address of the object itself, which is something that IDAPro knows nothing about. In contrast, VSA considers each malloc site m to be a "memory region" (consisting of the objects allocated at m), and the memory region for m serves as a representative for the base addresses of those objects. This lets ASI handle the use of an offset from an object's base address similar to the way that it handles a stack-frame offset—with the net result that ASI is able to capture information about the fine-grained structure of dynamically allocated objects. The object fields discovered in this way become a-locs for the next round of VSA, which will then discover an over-approximation of their contents.

ASI is complementary to VSA: ASI addresses only the issue of identifying the structure of aggregates, whereas VSA addresses the issue of (over-approximating) the contents of memory locations. ASI provides an improved method for the "variable-identification" facility of IDAPro, which uses only much cruder techniques (and only takes into account statically known memory addresses and stack offsets). Moreover, ASI requires more information to be on hand than is available in IDAPro (such as the sizes of arrays and iteration counts for loops). Fortunately, this is exactly the information that is available after VSA has been carried out, which means that ASI can be used in conjunction with VSA to obtain improved results: after a first round of VSA, the results of ASI are used to refine the a-loc abstraction, after which VSA is run again—generally producing more precise results.

3.2 CodeSurfer/x86

The value-sets for the a-locs at each program point are used to determine each point's sets of used, killed, and possibly-killed a-locs; these are emitted in a format that is suitable for input to CodeSurfer.

CodeSurfer is a tool for code understanding and code inspection that supports both a graphical user interface (GUI) and an API (as well as a scripting language) to provide access to a program's system dependence graph (SDG) [28], as well as other information stored in CodeSurfer's IRs.[7] An SDG consists of a set of program dependence graphs (PDGs), one for each procedure in the program. A vertex in a PDG corresponds to a construct in the program, such as an instruction, a call to a procedure, an actual parameter of a call, or a formal parameter of a procedure. The edges correspond to data and control dependences between the vertices [21]. The PDGs are connected together with interprocedural edges that represent control dependences between procedure calls and entries, data dependences between actual parameters and formal parameters, and data dependences between return values and receivers of return values.

Dependence graphs are invaluable for many applications, because they highlight chains of dependent instructions that may be widely scattered through the program. For example, given an instruction, it is often useful to know its *data-dependence predecessors* (instructions that write to locations read by that instruction) and its *control-dependence predecessors* (control points that may affect whether a given instruction gets executed). Similarly, it may be useful to know for a given instruction its *data-dependence successors* (instructions that read locations written by that instruction) and *control-dependence* successors (instructions whose execution depends on the decision made at a given control point).

CodeSurfer's GUI supports browsing ("surfing") of an SDG, along with a variety of operations for making queries about the SDG—such as slicing [28] and chopping [35].[8] The GUI allows a user to navigate through a program's source code using these dependences in a manner analogous to navigating the World Wide Web.

CodeSurfer's API provides a programmatic interface to these operations, as well as to lower-level information, such as the individual nodes and edges of the program's SDG, call graph, and control-flow graph, and a node's sets of used, killed, and possibly-killed a-locs. By writing programs that traverse CodeSurfer's IRs to implement additional program analyses, the API can be used to extend CodeSurfer's capabilities.

[7] In addition to the SDG, CodeSurfer's IRs include abstract-syntax trees, control-flow graphs (CFGs), a call graph, VSA results, the sets of used, killed, and possibly killed a-locs at each instruction, and information about the structure and layout of global memory, activation records, and dynamically allocated storage.

[8] A backward slice of a program with respect to a set of program points S is the set of all program points that might affect the computations performed at S; a forward slice with respect to S is the set of all program points that might be affected by the computations performed at members of S [28]. A program chop between a set of source program points S and a set of target program points T shows how S can affect the points in T [35]. Chopping is a key operation in information-flow analysis.

CodeSurfer/x86 provides some unique capabilities for answering an analyst's questions. For instance, given a worm, CodeSurfer/x86's analysis results have been used to obtain information about the worm's target-discovery, propagation, and activation mechanisms by

- locating sites of system calls,
- finding the instructions by which arguments are passed, and
- following dependences backwards from those instructions to identify where the values come from.

Because the techniques described in §3.1 are able to recover quite rich information about memory-access operations, the answers that CodeSurfer/x86 furnishes to such questions account for the movement of data through memory—not just the movement of data through registers, as in some prior work (e.g., [18, 11]).

3.3 Goals, Capabilities, and Assumptions

A few words are in order about the goals, capabilities, and assumptions underlying CodeSurfer/x86.

The constraint that symbol-table and debugging information are off-limits complicated the task of creating CodeSurfer/x86; however, the results of VSA and ASI provide a substitute for such information. This allowed us to create a tool that can be used when symbol-table and debugging information is absent or untrusted.

Given an executable as input, the goal is to check whether the executable conforms to a "standard" compilation model—i.e., a runtime stack is maintained; activation records (ARs) are pushed onto the stack on procedure entry and popped from the stack on procedure exit; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f; the program's instructions occupy a fixed area of memory, are not self-modifying, and are separate from the program's data. If the executable conforms to this model, CodeSurfer/x86 creates an IR for it. If it does not conform to the model, then one or more violations will be discovered, and corresponding error reports are issued.

The goal for CodeSurfer/x86 is to provide (i) a tool for security analysis, and (ii) a general infrastructure for additional analysis of executables. Thus, as a practical measure, when the system produces an error report, a choice is made about how to accommodate the error so that analysis can continue (i.e., the error is optimistically treated as a false positive), and an IR is produced; if the analyst can determine that the error report is indeed a false positive, then the IR is valid.

The analyzer does not care whether the program was compiled from a high-level language, or hand-written in assembly code. In fact, some pieces of the program may be the output from a compiler (or from multiple compilers, for different high-level languages), and others hand-written assembly code. Still, it is easiest to talk about the information that VSA and ASI are capable of recovering in terms of the features that a high-level programming language allows: VSA and ASI are capable of recovering information from programs that use global variables, local variables, pointers, structures, arrays, heap-allocated storage, pointer arithmetic, indirect jumps, recursive procedures, indirect calls through function pointers, virtual-function calls, and DLLs (but, at present, not run-time code generation or self-modifying code).

Compiler optimizations often make VSA and ASI *less* difficult, because more of the computation's critical data resides in registers, rather than in memory; register operations are more easily deciphered than memory operations.

The major assumption that we make about IDAPro is that it is able to disassemble a program and build an adequate collection of *preliminary* IRs for it. Even though (i) the CFG created by IDAPro may be incomplete due to indirect jumps, and (ii) the call-graph created by IDAPro may be incomplete due to indirect calls, incomplete IRs do *not* trigger error reports. Both the CFG and the call-graph are fleshed out according to information recovered during the course of VSA/ASI iteration. In fact, the relationship between VSA/ASI iteration and the preliminary IRs created by IDAPro is similar to the relationship between a points-to-analysis algorithm in a C compiler and the preliminary IRs created by the C compiler's front end. In both cases, the preliminary IRs are fleshed out during the course of analysis.

4 Model-Checking Facilities

Model checking [12] involves the use of sophisticated pattern-matching techniques to answer questions about the flow of execution in a program: a model of the program's possible behavior is created and checked for conformance with a model of expected behavior (as specified by a user query). In essence, model-checking algorithms explore the program's state-space and answer questions about whether a bad state can be reached during an execution of the program.

For model checking, the CodeSurfer/x86 IRs are used to build a *weighted pushdown system* (WPDS) [7, 36, 37, 32] that models possible program behaviors. WPDSs generalize a model-checking technology known as *pushdown systems* (PDSs) [6, 22], which have been used for software model checking in the Moped [39, 38] and MOPS [10] systems. Compared to ordinary (unweighted) PDSs, WPDSs are capable of representing more powerful kinds of abstractions of runtime states [37, 32], and hence go beyond the capabilities of PDSs. For instance, the use of WPDSs provides a way to address certain kinds of security-related queries that cannot be answered by MOPS.

WPDS++ [31] is a library that implements the symbolic algorithms from [37, 32] for solving WPDS reachability problems. We follow the standard approach of using a PDS to model the interprocedural CFG (one of CodeSurfer/x86's IRs). The stack symbols correspond to program locations; there is only a single PDS state; and PDS rules encode control flow as follows:

Rule	Control flow modeled
$q\langle u\rangle \hookrightarrow q\langle v\rangle$	Intraprocedural CFG edge $u \to v$
$q\langle c\rangle \hookrightarrow q\langle entry_P\ r\rangle$	Call to P from c that returns to r
$q\langle x\rangle \hookrightarrow q\langle\rangle$	Return from a procedure at exit node x

In a configuration of the PDS, the symbol at the top of the stack corresponds to the current program location, and the rest of the stack holds return-site locations—this allows the PDS to model the behavior of the program's runtime execution stack.

An encoding of the interprocedural CFG as a PDS is sufficient for answering queries about reachable control states (as the Path Inspector does; see below): the reachability algorithms of WPDS++ can determine if an undesirable PDS configuration is reachable.

However, WPDS++ also supports *weighted* PDSs, which are PDSs in which each rule is weighted with an element of a (user-defined) semiring. The use of weights allows WPDS++ to perform interprocedural dataflow analysis by using the semiring's *extend* operator to compute weights for sequences of rule firings and using the semiring's *combine* operator to take the meet of weights generated by different paths [37, 32]. (When the weights on rules are conservative abstract data transformers, an over-approximation to the set of reachable concrete configurations is obtained, which means that counterexamples reported by WPDS++ may actually be infeasible.)

The advantage of answering reachability queries on WPDSs over conventional dataflow-analysis methods is that the latter merge together the values for all states associated with the same program point, regardless of the states' calling context. With WPDSs, queries can be posed with respect to a regular language of stack configurations [7, 36, 37, 32]. (Conventional merged dataflow information can also be obtained [37].)

CodeSurfer/x86 can also be used in conjunction with GrammaTech's Path Inspector tool. The Path Inspector provides a user interface for automating safety queries that are only concerned with the possible control configurations that an executable can reach. The Path Inspector checks *sequencing properties* of events in a program, which can be used to answer such questions as "Is it possible for the program to bypass the authentication routine?" (which indicates that the program may contain a trapdoor), or "Can this *login* program bypass the code that writes to the log file?" (which indicates that the program may be a Trojan *login* program).

With the Path Inspector, such questions are posed as questions about the existence of problematic event sequences; after checking the query, if a problematic path exists, it is displayed in the Path Inspector tool. This lists all of the program points that may occur along the problematic path. These items are linked to the source code; the analyst can navigate from a point in the path to the corresponding source-code element. In addition, the Path Inspector allows the analyst to step forward and backward through the path, while simultaneously stepping through the source code. (The code-stepping operations are similar to the single-stepping operations in a traditional debugger.)

The Path Inspector uses an automaton-based approach to model checking: the query is specified as a finite automaton that captures forbidden sequences of program locations. This "query automaton" is combined with the program model (a WPDS) using a cross-product construction, and the reachability algorithms of WPDS++ are used to determine if an error configuration is reachable. If an error configuration is reachable, then *witnesses* (see [37]) can be used to produce a program path that drives the query automaton to an error state.

The Path Inspector includes a GUI for instantiating many common reachability queries [19], and for displaying counterexample paths in the disassembly listing. In the current implementation, transitions in the query automaton are triggered by program points that the user specifies either manually, or using result sets from CodeSurfer queries. Future versions of the Path Inspector will support more sophisticated queries in which transitions are triggered by matching an AST pattern against a program location, and query states can be instantiated based on pattern bindings.

5 Related Work

Previous work on analyzing memory accesses in executables has dealt with memory accesses very conservatively: generally, if a register is assigned a value from memory, it is assumed to take on any value. VSA does a much better job than previous work because it tracks the integer-valued and address-valued quantities that the program's data objects can hold; in particular, VSA tracks the values of data objects other than just the hardware registers, and thus is not forced to give up all precision when a load from memory is encountered.

The basic goal of the algorithm proposed by Debray et al. [18] is similar to that of VSA: for them, it is to find an over-approximation of the set of values that each *register* can hold at each program point; for us, it is to find an over-approximation of the set of values that each (abstract) data object can hold at each program point, where data objects include *memory locations* in addition to registers. In their analysis, a set of addresses is approximated by a set of congruence values: they keep track of only the low-order bits of addresses. However, unlike VSA, their algorithm does not make any effort to track values that are not in registers. Consequently, they lose a great deal of precision whenever there is a load from memory.

Cifuentes and Fraboulet [11] give an algorithm to identify an intraprocedural slice of an executable by following the program's use-def chains. However, their algorithm also makes no attempt to track values that are not in registers, and hence cuts short the slice when a load from memory is encountered.

The two pieces of work that are most closely related to VSA are the algorithm for data-dependence analysis of assembly code of Amme et al. [2] and the algorithm for pointer analysis on a low-level intermediate representation of Guo et al. [24]. The algorithm of Amme et al. performs only an *intra*procedural analysis, and it is not clear whether the algorithm fully accounts for dependences between memory locations. The algorithm of Guo et al. [24] is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [42] to achieve context-sensitivity. The transfer functions are parameterized by "unknown initial values" (UIVs); however, it is not clear whether the the algorithm accounts for the possibility of called procedures corrupting the memory locations that the UIVs represent.

References

1. PREfast with driver-specific rules, October 2004. WHDC, Microsoft Corp., http://www.microsoft.com/whdc/devtools/tools/PREfast-drv.mspx.
2. W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *Int. J. Parallel Proc.*, 2000.
3. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.
4. G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *IFIP Working Conf. on Verified Software: Theories, Tools, Experiments*, 2005.
5. T. Ball and S.K. Rajamani. The SLAM toolkit. In *Computer Aided Verif.*, volume 2102 of *Lec. Notes in Comp. Sci.*, pages 260–264, 2001.

6. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *Proc. CONCUR*, volume 1243 of *Lec. Notes in Comp. Sci.*, pages 135–150. Springer-Verlag, 1997.
7. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Princ. of Prog. Lang.*, pages 62–73, 2003.
8. W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice&Experience*, 30:775–802, 2000.
9. H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code. In *Network and Dist. Syst. Security*, 2004.
10. H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Conf. on Comp. and Commun. Sec.*, pages 235–244, November 2002.
11. C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Int. Conf. on Softw. Maint.*, pages 188–195, 1997.
12. E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. The M.I.T. Press, 1999.
13. CodeSurfer, GrammaTech, Inc., http://www.grammatech.com/products/codesurfer/.
14. J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Int. Conf. on Softw. Eng.*, pages 439–448, 2000.
15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Princ. of Prog. Lang.*, pages 238–252, 1977.
16. D.S. Coutant, S. Meloy, and M. Ruscetta. DOC: A practical approach to source-level debugging of globally optimized code. In *Prog. Lang. Design and Impl.*, 1988.
17. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Prog. Lang. Design and Impl.*, pages 57–68, New York, NY, 2002. ACM Press.
18. S.K. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Princ. of Prog. Lang.*, pages 12–24, 1998.
19. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Int. Conf. on Softw. Eng.*, 1999.
20. D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Op. Syst. Design and Impl.*, pages 1–16, 2000.
21. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *Trans. on Prog. Lang. and Syst.*, 3(9):319–349, 1987.
22. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
23. Fast Library Identification and Recognition Technology, DataRescue sa/nv, Liège, Belgium, http://www.datarescue.com/idabase/flirt.htm.
24. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd Int. Symp. on Code Gen. and Opt.*, pages 291–302, 2005.
25. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Softw. Tools for Tech. Transfer*, 2(4), 2000.
26. J.L. Hennessy. Symbolic debugging of optimized code. *Trans. on Prog. Lang. and Syst.*, 4(3):323–344, 1982.
27. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Princ. of Prog. Lang.*, pages 58–70, 2002.
28. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Trans. on Prog. Lang. and Syst.*, 12(1):26–60, January 1990.
29. M. Howard. Some bad news and some good news. October 2002. MSDN, Microsoft Corp., http://msdn.microsoft.com/library/default.asp?url=/library/en-

us/dncode/html/secure10102002.asp.
30. IDAPro disassembler, http://www.datarescue.com/idabase/.
31. N. Kidd, T. Reps, D. Melski, and A. Lal. WPDS++: A C++ library for weighted pushdown systems, 2004. http://www.cs.wisc.edu/wpis/wpds++/.
32. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *Computer Aided Verif.*, 2005.
33. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *European Symp. on Programming*, 2005.
34. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*, pages 119–132, 1999.
35. T. Reps and G. Rosay. Precise interprocedural chopping. In *Found. of Softw. Eng.*, 1995.
36. T. Reps, S. Schwoon, and S. Jha. Weighted pushdown systems and their application to interprocedural datafbw analysis. In *Static Analysis Symp.*, 2003.
37. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural datafbw analysis. *Sci. of Comp. Prog.* To appear.
38. S. Schwoon. Moped system. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
39. S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical Univ. of Munich, Munich, Germany, July 2002.
40. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Dist. Syst. Security*, February 2000.
41. D.W. Wall. Systems for late code modification. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 275–293. Springer-Verlag, 1992.
42. R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Prog. Lang. Design and Impl.*, pages 1–12, 1995.
43. P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs*. PhD thesis, Univ. of California, Berkeley, 1984.